4

Abstract Syntax

ab-stract: disassociated from any specific instance

Webster's Dictionary

A compiler must do more than recognize whether a sentence belongs to the language of a grammar – it must do something useful with that sentence. The *semantic actions* of a parser can do useful things with the phrases that are parsed.

In a recursive-descent parser, semantic action code is interspersed with the control flow of the parsing actions. In a parser specified in JavaCC, semantic actions are fragments of Java program code attached to grammar productions. SableCC, on the other hand, automatically generates syntax trees as it parses.

4.1 SEMANTIC ACTIONS

Each terminal and nonterminal may be associated with its own type of semantic value. For example, in a simple calculator using Grammar 3.37, the type associated with `exp` and `INT` might be `int`; the other tokens would not need to carry a value. The type associated with a token must, of course, match the type that the lexer returns with that token.

For a rule $A \rightarrow B\ C\ D$, the semantic action must return a value whose type is the one associated with the nonterminal A. But it can build this value from the values associated with the matched terminals and nonterminals B, C, D.

RECURSIVE DESCENT

In a recursive-descent parser, the semantic actions are the values returned by parsing functions, or the side effects of those functions, or both. For each ter-

```
class Token {int kind; Object val;
             Token(int k, Object v) {kind=k; val=v;}
            }
final int EOF=0, ID=1, NUM=2, PLUS=3, MINUS=4,  ···

int lookup(String id) {  ···  }

int F_follow[] = { PLUS, TIMES, RPAREN, EOF };

int F() {switch (tok.kind) {
         case ID:   int i=lookup((String)(tok.val)); advance(); return i;
         case NUM:  int i=((Integer)(tok.val)).intValue();
                    advance(); return i;
         case LPAREN:  eat(LPAREN);
                       int i = E();
                       eatOrSkipTo(RPAREN, F_follow);
                       return i;
         case EOF:
         default:   print("expected ID, NUM, or left-paren");
                    skipto(F_follow); return 0;
        }}

int T_follow[] = { PLUS, RPAREN, EOF };

int T() {switch (tok.kind) {
         case ID:
         case NUM:
         case LPAREN: return Tprime(F());
         default: print("expected ID, NUM, or left-paren");
                  skipto(T_follow);
                  return 0;
        }}

int Tprime(int a)  {switch (tok.kind) {
        case TIMES: eat(TIMES); return Tprime(a*F());
        case PLUS:
        case RPAREN:
        case EOF:  return a;
        default:  ···
       }}

void eatOrSkipTo(int expected, int[] stop) {
   if (tok.kind==expected)
        eat(expected);
   else {print(···); skipto(stop);}
}
```

PROGRAM 4.1. Recursive-descent interpreter for part of Grammar 3.15.

```
void Start() :
{ int i; }
{ i=Exp() <EOF>  { System.out.println(i); }
}
int Exp() :
{ int a,i; }
{ a=Term()
  ( "+" i=Term() { a=a+i; }
  | "-" i=Term() { a=a-i; }
  )*
  { return a; }
}
int Term() :
{ int a,i; }
{ a=Factor()
  ( "*" i=Factor() { a=a*i; }
  | "/" i=Factor() { a=a/i; }
  )*
  { return a; }
}
int Factor() :
{ Token t; int i; }
{ t=<IDENTIFIER>      { return lookup(t.image); }
| t=<INTEGER_LITERAL> { return Integer.parseInt(t.image); }
| "(" i=Exp() ")"     { return i; }
}
```

PROGRAM 4.2. JavaCC version of a variant of Grammar 3.15.

minal and nonterminal symbol, we associate a *type* (from the implementation language of the compiler) of *semantic values* representing phrases derived from that symbol.

Program 4.1 is a recursive-descent interpreter for part of Grammar 3.15. The tokens `ID` and `NUM` must now carry values of type `string` and `int`, respectively. We will assume there is a lookup table mapping identifiers to integers. The type associated with E, T, F, etc., is `int`, and the semantic actions are easy to implement.

The semantic action for an artificial symbol such as T' (introduced in the elimination of left recursion) is a bit tricky. Had the production been $T \rightarrow T * F$, then the semantic action would have been

```
int a = T(); eat(TIMES); int b=F(); return a*b;
```

With the rearrangement of the grammar, the production $T' \rightarrow *FT'$ is missing the left operand of the $*$. One solution is for T to pass the left operand as an argument to T', as shown in Program 4.1.

AUTOMATICALLY GENERATED PARSERS

A parser specification for JavaCC consists of a set of grammar rules, each annotated with a semantic action that is a Java statement. Whenever the generated parser reduces by a rule, it will execute the corresponding semantic action fragment.

Program 4.2 shows how this works for a variant of Grammar 3.15. Every `INTEGER_CONSTANT` terminal and every nonterminal (except `Start`) carries a value. To access this value, give the terminal or nonterminal a name in the grammar rule (such as `i` in Program 4.2), and access this name as a variable in the semantic action.

SableCC, unlike JavaCC, has no way to attach action code to productions. However, SableCC automatically generates syntax tree classes, and a parser generated by SableCC will build syntax trees using those classes. For JavaCC, there are several companion tools, including JJTree and JTB (the Java Tree Builder), which, like SableCC, generate syntax tree classes and insert action code into the grammar for building syntax trees.

4.2 ABSTRACT PARSE TREES

It is possible to write an entire compiler that fits within the semantic action phrases of a JavaCC or SableCC parser. However, such a compiler is difficult to read and maintain, and this approach constrains the compiler to analyze the program in exactly the order it is parsed.

To improve modularity, it is better to separate issues of syntax (parsing) from issues of semantics (type-checking and translation to machine code). One way to do this is for the parser to produce a *parse tree* – a data structure that later phases of the compiler can traverse. Technically, a parse tree has exactly one leaf for each token of the input and one internal node for each grammar rule reduced during the parse.

Such a parse tree, which we will call a *concrete parse tree,* representing the *concrete syntax* of the source language, may be inconvenient to use directly. Many of the punctuation tokens are redundant and convey no information – they are useful in the input string, but once the parse tree is built, the structure

$E \to E + E$
$E \to E - E$
$E \to E * E$
$E \to E / E$
$E \to \text{id}$
$E \to \text{num}$

GRAMMAR 4.3. Abstract syntax of expressions.

of the tree conveys the structuring information more conveniently.

Furthermore, the structure of the parse tree may depend too much on the grammar! The grammar transformations shown in Chapter 3 – factoring, elimination of left recursion, elimination of ambiguity – involve the introduction of extra nonterminal symbols and extra grammar productions for technical purposes. These details should be confined to the parsing phase and should not clutter the semantic analysis.

An *abstract syntax* makes a clean interface between the parser and the later phases of a compiler (or, in fact, for the later phases of other kinds of program-analysis tools such as dependency analyzers). The abstract syntax tree conveys the phrase structure of the source program, with all parsing issues resolved but without any semantic interpretation.

Many early compilers did not use an abstract syntax data structure because early computers did not have enough memory to represent an entire compilation unit's syntax tree. Modern computers rarely have this problem. And many modern programming languages (ML, Modula-3, Java) allow forward reference to identifiers defined later in the same module; using an abstract syntax tree makes compilation easier for these languages. It may be that Pascal and C require clumsy *forward* declarations because their designers wanted to avoid an extra compiler pass on the machines of the 1970s.

Grammar 4.3 shows an abstract syntax of the expression language is Grammar 3.15. This grammar is completely impractical for parsing: The grammar is quite ambiguous, since precedence of the operators is not specified.

However, Grammar 4.3 is not meant for parsing. The parser uses the *concrete syntax* to build a parse tree for the *abstract syntax*. The semantic analysis phase takes this *abstract syntax tree*; it is not bothered by the ambiguity of the grammar, since it already has the parse tree!

The compiler will need to represent and manipulate abstract syntax trees as

```
Exp Start() :
  { Exp e; }
  { e=Exp() { return e; }
  }
Exp Exp() :
  { Exp e1,e2; }
  { e1=Term()
      ( "+" e2=Term() { e1=new  PlusExp(e1,e2); }
      | "-" e2=Term() { e1=new MinusExp(e1,e2); }
      )*
    { return e1; }
  }
Exp Term() :
  { Exp e1,e2; }
  { e1=Factor()
      ( "*" e2=Factor() { e1=new  TimesExp(e1,e2); }
      | "/" e2=Factor() { e1=new DivideExp(e1,e2); }
      )*
    { return e1; }
  }
Exp Factor() :
  { Token t; Exp e; }
  { ( t=<IDENTIFIER>      { return new Identifier(t.image); } |
      t=<INTEGER_LITERAL> { return new IntegerLiteral(t.image); } |
      "(" e=Exp() ")"     { return e; } )
  }
```

PROGRAM 4.4. Building syntax trees for expressions.

data structures. In Java, these data structures are organized according to the principles outlined in Section 1.3: an abstract class for each nonterminal, a subclass for each production, and so on. In fact, the classes of Program 4.5 are abstract syntax classes for Grammar 4.3. An alternate arrangement, with all the different binary operators grouped into an `OpExp` class, is also possible.

Let us write an interpreter for the expression language in Grammar 3.15 by first building syntax trees and then interpreting those trees. Program 4.4 is a JavaCC grammar with semantic actions that produce syntax trees. Each class of syntax-tree nodes contains an `eval` function; when called, such a function will return the value of the represented expression.

POSITIONS

In a one-pass compiler, lexical analysis, parsing, and semantic analysis (type-checking) are all done simultaneously. If there is a type error that must be reported to the user, the *current* position of the lexical analyzer is a reason-

```
public abstract class Exp {
   public abstract int eval();
}
public class PlusExp extends Exp {
   private Exp e1,e2;
   public PlusExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
   public int eval() {
       return e1.eval()+e2.eval();
   }
}
public class MinusExp extends Exp {
   private Exp e1,e2;
   public MinusExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
   public int eval() {
       return e1.eval()-e2.eval();
   }
}
public class TimesExp extends Exp {
   private Exp e1,e2;
   public TimesExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
   public int eval() {
       return e1.eval()*e2.eval();
   }
}
public class DivideExp extends Exp {
   private Exp e1,e2;
   public DivideExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
   public int eval() {
       return e1.eval()/e2.eval();
   }
}
public class Identifier extends Exp {
   private String f0;
   public Identifier(String n0) { f0 = n0; }
   public int eval() {
       return lookup(f0);
   }
}
public class IntegerLiteral extends Exp {
   private String f0;
   public IntegerLiteral(String n0) { f0 = n0; }
   public int eval() {
       return Integer.parseInt(f0);
   }
}
```

PROGRAM 4.5. Exp class for Program 4.4.

able approximation of the source position of the error. In such a compiler, the lexical analyzer keeps a "current position" global variable, and the error-message routine just prints the value of that variable with each message.

A compiler that uses abstract-syntax-tree data structures need not do all the parsing and semantic analysis in one pass. This makes life easier in many ways, but slightly complicates the production of semantic error messages. The lexer reaches the end of file before semantic analysis even begins; so if a semantic error is detected in traversing the abstract syntax tree, the *current* position of the lexer (at end of file) will not be useful in generating a line number for the error message. Thus, the source-file position of each node of the abstract syntax tree must be remembered, in case that node turns out to contain a semantic error.

To remember positions accurately, the abstract-syntax data structures must be sprinkled with `pos` fields. These indicate the position, within the original source file, of the characters from which these abstract-syntax structures were derived. Then the type-checker can produce useful error messages. (The syntax constructors we will show in Figure 4.9 do not have `pos` fields; any compiler that uses these exactly as given will have a hard time producing accurately located error messages.)

The lexer must pass the source-file positions of the beginning and end of each token to the parser. We can augment the types `Exp`, etc. with a `position` field; then each constructor must take a `pos` argument to initialize this field. The positions of leaf nodes of the syntax tree can be obtained from the tokens returned by the lexical analyzer; internal-node positions can be derived from the positions of their subtrees. This is tedious but straightforward.

4.3 VISITORS

Each abstract syntax class of Program 4.5 has a constructor for building syntax trees, and an `eval` method for returning the value of the represented expression. This is an *object-oriented* style of programming. Let us consider an alternative.

Suppose the code for evaluating expressions is written *separately* from the abstract syntax classes. We might do that by examining the syntax-tree data structure by using `instanceof` and by fetching public class variables that represent subtrees. This is a *syntax separate from interpretations* style of programming.

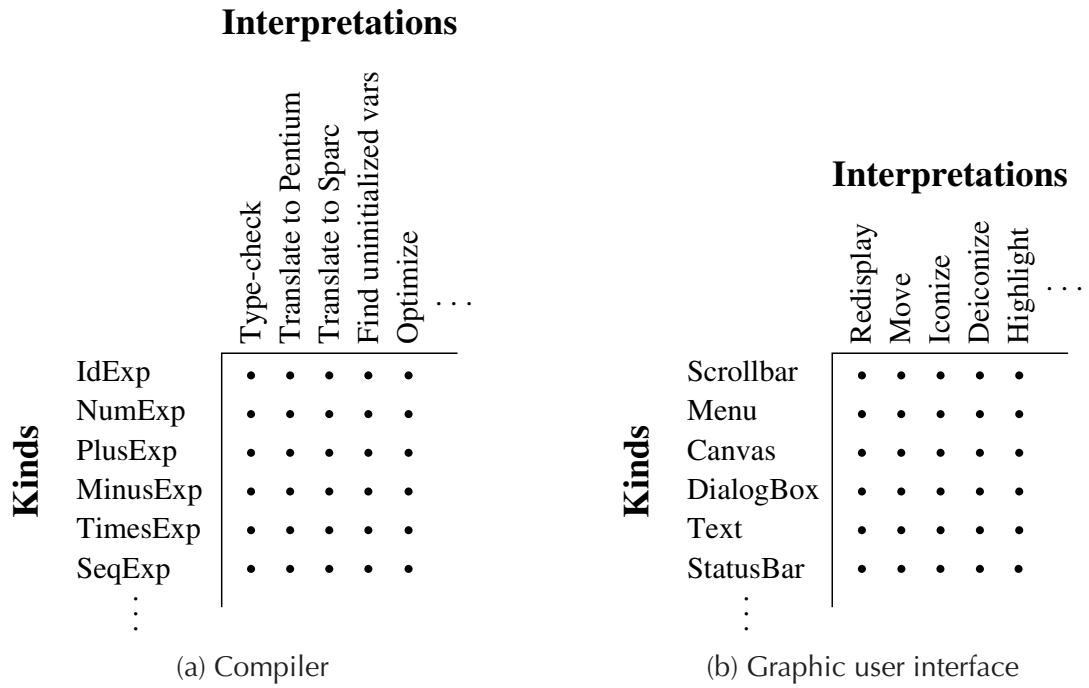

FIGURE 4.6. Orthogonal directions of modularity.

The choice of style affects the modularity of the compiler. In a situation such as this, we have several *kinds* of objects: compound statements, assignment statements, print statements, and so on. And we also may have several different *interpretations* of these objects: type-check, translate to Pentium code, translate to Sparc code, optimize, interpret, and so on.

Each *interpretation* must be applied to each *kind*; if we add a new kind, we must implement each interpretation for it; and if we add a new interpretation, we must implement it for each kind. Figure 4.6 illustrates the orthogonality of kinds and interpretations – for compilers, and for graphic user interfaces, where the *kinds* are different widgets and gadgets, and the *interpretations* are move, hide, and redisplay commands.

If the *syntax separate from interpretations* style is used, then it is easy and modular to add a new *interpretation:* One new function is written, with clauses for the different kinds all grouped logically together. On the other hand, it will not be modular to add a new *kind*, since a new clause must be added to every interpretation function.

With the *object-oriented* style, each interpretation is just a *method* in all the classes. It is easy and modular to add a new *kind:* All the interpretations of that kind are grouped together as methods of the new class. But it is not

```
public abstract class Exp {
   public abstract int accept(Visitor v);
}
public class PlusExp extends Exp {
   public Exp e1,e2;
   public PlusExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
   public int accept(Visitor v) {
       return v.visit(this);
   }
}
public class MinusExp extends Exp {
   public Exp e1,e2;
   public MinusExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
   public int accept(Visitor v) {
       return v.visit(this);
   }
}
public class TimesExp extends Exp {
   public Exp e1,e2;
   public TimesExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
   public int accept(Visitor v) {
       return v.visit(this);
   }
}
public class DivideExp extends Exp {
   public Exp e1,e2;
   public DivideExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
   public int accept(Visitor v) {
       return v.visit(this);
   }
}
public class Identifier extends Exp {
   public String f0;
   public Identifier(String n0) { f0 = n0; }
   public int accept(Visitor v) {
       return v.visit(this);
   }
}
public class IntegerLiteral extends Exp {
   public String f0;
   public IntegerLiteral(String n0) { f0 = n0; }
   public int accept() {
       return v.visit(this);
   }
}
```

PROGRAM 4.7. Syntax classes with accept methods.

```
public interface Visitor {
   public int visit(PlusExp n);
   public int visit(MinusExp n);
   public int visit(TimesExp n);
   public int visit(DivideExp n);
   public int visit(Identifier n);
   public int visit(IntegerLiteral n);
}

public class Interpreter implements Visitor {
   public int visit(PlusExp n) {
       return n.e1.accept(this)+n.e2.accept(this);
   }
   public int visit(MinusExp n) {
       return n.e1.accept(this)-n.e2.accept(this);
   }
   public int visit(TimesExp n) {
       return n.e1.accept(this)*n.e2.accept(this);
   }
   public int visit(DivideExp n) {
       return n.e1.accept(this)/n.e2.accept(this);
   }
   public int visit(Identifier n) {
       return lookup(n.f0);
   }
   public int visit(IntegerLiteral n) {
       return Integer.parseInt(n.f0);
   }
}
```

PROGRAM 4.8. An interpreter visitor.

modular to add a new *interpretation:* A new method must be added to every class.

For graphic user interfaces, each application will want to make its own kinds of widgets; it is impossible to predetermine one set of widgets for everyone to use. On the other hand, the set of common operations (interpretations) is fixed: The window manager demands that each widget support only a certain interface. Thus, the *object-oriented* style works well, and the *syntax separate from interpretations* style would not be as modular.

For programming languages, on the other hand, it works very well to fix a syntax and then provide many interpretations of that syntax. If we have a compiler where one interpretation is *translate to Pentium* and we wish to port that compiler to the Sparc, then not only must we add operations for generat-

ing Sparc code but we might also want to remove (in this configuration) the Pentium code-generation functions. This would be very inconvenient in the object-oriented style, requiring each class to be edited. In the *syntax separate from interpretations* style, such a change is modular: We remove a Pentium-related module and add a Sparc module.

We prefer a syntax-separate-from-interpretations style. Fortunately, we can use this style without employing `instanceof` expressions for accessing syntax trees. Instead, we can use a technique known as the Visitor pattern. A visitor implements an interpretation; it is an object which contains a `visit` method for each syntax-tree class. Each syntax-tree class should contain an `accept` method. An `accept` method serves as a hook for all interpretations. It is called by a visitor and it has just one task: It passes control back to an appropriate method of the visitor. Thus, control goes back and forth between a visitor and the syntax-tree classes.

Intuitively, the visitor calls the `accept` method of a node and asks "what is your class?" The `accept` method answers by calling the corresponding `visit` method of the visitor. Code for the running example, using visitors, is given in Programs 4.7 and 4.8. Every visitor implements the interface `Visitor`. Notice that each `accept` method takes a visitor as an argument, and that each `visit` method takes a syntax-tree-node object as an argument.

In Programs 4.7 and 4.8, the `visit` and `accept` methods all return `int`. Suppose we want instead to return `String`. In that case, we can add an appropriate `accept` method to each syntax tree class, and we can write a new visitor class in which all `visit` methods return `String`.

The main difference between the object-oriented style and the syntax-separate-from-interpretations style is that, for example, the interpreter code in Program 4.5 is in the `eval` methods while in Program 4.8 it is in the `Interpreter` visitor.

In summary, with the Visitor pattern we can add a new interpretation without editing and recompiling existing classes, provided that each of the appropriate classes has an `accept` method. The following table summarizes some advantages of the Visitor pattern:

	Frequent type casts?	Frequent recompilation?
Instanceof and type casts	Yes	No
Dedicated methods	No	Yes
The Visitor pattern	No	No

ABSTRACT SYNTAX FOR MiniJava

Figure 4.9 shows classes for the abstract syntax of MiniJava. The meaning of each constructor in the abstract syntax should be clear after a careful study of Appendix A, but there are a few points that merit explanation.

Only the constructors are shown in Figure 4.9; the object field variables correspond exactly to the names of the constructor arguments. Each of the six list classes is implemented in the same way, for example:

```
public class ExpList {
   private Vector list;
   public ExpList() {
      list = new Vector();
   }
   public void addElement(Exp n) {
      list.addElement(n);
   }
   public Exp elementAt(int i)  {
      return (Exp)list.elementAt(i);
   }
   public int size() {
      return list.size();
   }
}
```

Each of the nonlist classes has an accept method for use with the visitor pattern. The interface `Visitor` is shown in Program 4.10.

We can construct a syntax tree by using nested `new` expressions. For example, we can build a syntax tree for the MiniJava statement:

```
x = y.m(1,4+5);
```

using the following Java code:

```
ExpList el = new ExpList();
el.addElement(new IntegerLiteral(1));
el.addElement(new Plus(new IntegerLiteral(4),
                       new IntegerLiteral(5)));
Statement s = new Assign(new Identifier("x"),
                         new Call(new IdentifierExp("y"),
                                  new Identifier("m"),
                                  el));
```

SableCC enables automatic generation of code for syntax tree classes, code for building syntax trees, and code for template visitors. For JavaCC, a companion tool called the Java Tree Builder (JTB) enables the generation of sim-

```
package syntaxtree;

Program(MainClass m, ClassDeclList cl)
MainClass(Identifier i1, Identifier i2, Statement s)

abstract class ClassDecl
ClassDeclSimple(Identifier i, VarDeclList vl, MethodDeclList ml)
ClassDeclExtends(Identifier i, Identifier j,
                         VarDeclList vl, MethodDeclList ml)  see Ch.14

VarDecl(Type t, Identifier i)
MethodDecl(Type t, Identifier i, FormalList fl, VarDeclList vl,
                   StatementList sl, Exp e)
Formal(Type t, Identifier i)

abstract class Type
IntArrayType() BooleanType() IntegerType() IdentifierType(String s)

abstract class Statement
Block(StatementList sl)
If(Exp e, Statement s1, Statement s2)
While(Exp e, Statement s)
Print(Exp e)
Assign(Identifier i, Exp e)
ArrayAssign(Identifier i, Exp e1, Exp e2)

abstract class Exp
And(Exp e1, Exp e2)
LessThan(Exp e1, Exp e2)
Plus(Exp e1, Exp e2) Minus(Exp e1, Exp e2) Times(Exp e1, Exp e2)
ArrayLookup(Exp e1, Exp e2)
ArrayLength(Exp e)
Call(Exp e, Identifier i, ExpList el)
IntegerLiteral(int i)
True()
False()
IdentifierExp(String s)
This()
NewArray(Exp e)
NewObject(Identifier i)
Not(Exp e)

Identifier(String s)

 list classes
ClassDeclList() ExpList() FormalList() MethodDeclList() StatementList() VarDeclList()
```

FIGURE 4.9. Abstract syntax for the MiniJava language.

```
public interface Visitor {
  public void visit(Program n);
  public void visit(MainClass n);
  public void visit(ClassDeclSimple n);
  public void visit(ClassDeclExtends n);
  public void visit(VarDecl n);
  public void visit(MethodDecl n);
  public void visit(Formal n);
  public void visit(IntArrayType n);
  public void visit(BooleanType n);
  public void visit(IntegerType n);
  public void visit(IdentifierType n);
  public void visit(Block n);
  public void visit(If n);
  public void visit(While n);
  public void visit(Print n);
  public void visit(Assign n);
  public void visit(ArrayAssign n);
  public void visit(And n);
  public void visit(LessThan n);
  public void visit(Plus n);
  public void visit(Minus n);
  public void visit(Times n);
  public void visit(ArrayLookup n);
  public void visit(ArrayLength n);
  public void visit(Call n);
  public void visit(IntegerLiteral n);
  public void visit(True n);
  public void visit(False n);
  public void visit(IdentifierExp n);
  public void visit(This n);
  public void visit(NewArray n);
  public void visit(NewObject n);
  public void visit(Not n);
  public void visit(Identifier n);
}
```

PROGRAM 4.10. MiniJava visitor

ilar code. The advantage of using such tools is that once the grammar is written, one can go straight on to writing visitors that operate on syntax trees. The disadvantage is that the syntax trees supported by the generated code may be less abstract than one could desire.

PROGRAM ABSTRACT SYNTAX

Add semantic actions to your parser to produce abstract syntax for the MiniJava language. Syntax-tree classes are available in `$MINIJAVA/chap4`, together with a `PrettyPrintVisitor`. If you use JavaCC, you can use JTB to generate the needed code automatically. Similarly, with SableCC, the needed code can be generated automatically.

FURTHER READING

Many compilers mix recursive-descent parsing code with semantic-action code, as shown in Program 4.1; Gries [1971] and Fraser and Hanson [1995] are ancient and modern examples. Machine-generated parsers with semantic actions (in special-purpose "semantic-action mini-languages") attached to the grammar productions were tried out in 1960s [Feldman and Gries 1968]; Yacc [Johnson 1975] was one of the first to permit semantic action fragments to be written in a conventional, general-purpose programming language.

The notion of *abstract syntax* is due to McCarthy [1963], who designed the abstract syntax for Lisp [McCarthy et al. 1962]. The abstract syntax was intended to be used for writing programs until designers could get around to creating a concrete syntax with human-readable punctuation (instead of **L**ots of **I**rritating **S**illy **P**arentheses), but programmers soon got used to programming directly in abstract syntax.

The search for a theory of programming-language semantics, and a notation for expressing semantics in a compiler-compiler, led to ideas such as *denotational semantics* [Stoy 1977]. The semantic interpreter shown in Programs 4.4 and 4.5 is inspired by ideas from denotational semantics, as is the idea of separating concrete syntax from semantics using the abstract syntax as a clean interface.

EXERCISES

4.1 Write a package of Java classes to express the abstract syntax of regular expressions.

4.2 Extend Grammar 3.15 such that a program is a sequence of either assignment statements or print statements. Each assignment statement assigns an expression

to an implicitly-declared variable; each print statement prints the value of an expression. Extend the interpreter in Program 4.1 to handle the new language.

4.3 Write a JavaCC version of the grammar from Exercise 4.2. Insert Java code for interpreting programs, in the style of Program 4.2.

4.4 Modify the JavaCC grammar from Exercise 4.3 to contain Java code for building syntax trees, in the style of Program 4.4. Write two interpreters for the language: one in object-oriented style and one that uses visitors.

4.5 In `$MINIJAVA/chap4/handcrafted/visitor`, there is a file with a visitor `PrettyPrintVisitor.java` for pretty printing syntax trees. Improve the pretty printing of nested `if` and `while` statements.

4.6 The visitor pattern in Program 4.7 has `accept` methods that return `int`. If one wanted to write some visitors that return integers, others that return class *A*, and yet others that return class *B*, one could modify all the classes in Program 4.7 to add two more `accept` methods, but this would not be very modular. Another way is to make the visitor return `Object` and cast each result, but this loses the benefit of compile-time type-checking. But there is a third way.

Modify Program 4.7 so that all the `accept` methods return `void`, and write two extensions of the `Visitor` class: one that computes an `int` for each `Exp`, and the other that computes a `float` for each `Exp`. Since the `accept` method will return `void`, the visitor object must have an instance variable into which each `accept` method can place its result. Explain why, if one then wanted to write a visitor that computed an object of class *C* for each `Exp`, no more modification of the `Exp` subclasses would be necessary.